MLIRSmith: Random Program Generation for Fuzzing MLIR Compiler Infrastructure

Haoyu Wang
College of Intelligence and Computing
Tianjin University
Tianjin, China
haoyuwang@tju.edu.cn

Junjie Chen†
College of Intelligence and Computing
Tianjin University
Tianjin, China
junjiechen@tju.edu.cn

Chuyue Xie
College of Intelligence and Computing
Tianjin University
Tianjin, China
xiechuyue@tju.edu.cn

Shuang Liu
College of Intelligence and Computing
Tianjin University
Tianjin, China
shuang.liu@tju.edu.cn

Zan Wang
College of Intelligence and Computing
Tianjin University
Tianjin, China
wangzan@tju.edu.cn

Qingchao Shen
College of Intelligence and Computing
Tianjin University
Tianjin, China
qingchao@tju.edu.cn

Yingquan Zhao
College of Intelligence and Computing
Tianjin University
Tianjin, China
zhaoyingquan@tju.edu.cn

***Abstract*—MLIR (Multi-Level Intermediate Representation) compiler infrastructure has gained popularity in recent years to support the construction of many compilers. Instead of designing a new IR with a single abstraction for each domain, MLIR compiler infrastructure provides systematic passes to support a wide range of functionalities for benefiting multiple domains together and introduces dialects to support different levels of abstraction in MLIR. Due to its fundamental role in compiler community, ensuring its quality is very critical. In this work, we propose MLIRSmith, the first fuzzing technique for MLIR compiler infrastructure. MLIRSmith employs a two-phase strategy to generate valid and diverse MLIR programs, which first constructs diverse program templates guided by extended MLIR syntax rules and then generates valid MLIR programs through template instantiation guided by our designed context-sensitive grammar. After applying MLIRSmith to the latest revision of MLIR compiler infrastructure, we detected 53 previously unknown bugs, among which 49/38 have been confirmed/fixed by developers. We also transform the high-level programs generated by NNSmith (a high-level program generator for deep learning compilers) to MLIR programs for indirectly fuzzing MLIR compiler infrastructure. During the same testing time, MLIRSmith largely outperforms such an indirect technique by detecting 328.57% more bugs and covering 194.67%/225.87% more lines/branches in MLIR compiler infrastructure.**

***Index Terms*—Test Program Generation, Compiler Fuzzing, MLIR Compiler Infrastructure**

I. Introduction

Compilers are one of the most fundamental software systems, which receive high-level source programs and output semantically equivalent low-level programs. Intermediate Representation (IR) is the key of compilers to facilitate the translation process [1]. Due to the existence of various domain-specific problems (e.g., language-specific optimizations), many compilers implemented their own IR (e.g., Relay IR for the deep learning compiler TVM [2]). However, designing and implementing individual IR with a single abstraction for each domain or each distinct requirement in a domain is quite costly and can be error-prone. In fact, different IRs often involve a lot of common problems, which can lead to substantial duplicate effort being wasted. Moreover, different tasks may be suitable for different abstraction levels of IRs, and the transformation among various IRs is also difficult. Under such circumstances, MLIR (Multi-Level IR) compiler infrastructure is proposed [3].

MLIR compiler infrastructure provides a wide range of common infrastructure to benefit multiple domains simultaneously and also introduces dialects to support multi-level IRs and facilitate their transformations [3]. Indeed, it has received extensive attention from both academia and industry rapidly. It not only promoted a great deal of research work [4], [5], [6], [7], [8], but also powered many compilers targeting different domains, such as the FORTRAN compiler Flang [9] and the deep learning compiler IREE [10]. Due to the fundamental role of MLIR compiler infrastructure, it is critical to ensure its correctness. Specifically, various domain-specific compilers are built on top of MLIR compiler infrastructure, and thus its bugs could cause its powered compilers to produce unexpected behaviors. That is, the bugs in MLIR compiler infrastructure have a wider impact than the bugs in an individual compiler, and the perniciousness of the latter has been clearly demonstrated by a lot of existing studies [11], [12]. Therefore, the significance of testing MLIR compiler infrastructure is self-evident in practice.

MLIR has its distinct characteristics, e.g., it uses dialects to manage multi-level IRs in the infrastructure and has its own data structure and semantics (such as regions that group a sequence of operations in a nested hierarchy) [3]. These characteristics make existing compiler test program generators not applicable to the MLIR compiler infrastructure since they are designed to either generate high-level source programs that

†Junjie Chen is the corresponding author.

serve as test inputs of a certain compiler (rather than the general compiler infrastructure) [12], [13], [14], [15] or generate domain-specific IRs [16], [17], [18] that do not share data structure and semantics with MLIR. In fact, high-level source programs for the compilers powered by the MLIR compiler infrastructure can be adopted to fuzz it by transforming them into MLIR programs via corresponding frontends. That is, some high-level program generators can be adopted to fuzz the MLIR compiler infrastructure, such as NNSmith that generates computation graphs for deep learning compilers [14]. However, these generators are constructed based on the characteristics of the target high-level programming languages rather than MLIR, and thus such a fuzzing method is indirect, which can limit MLIR-program diversity and negatively affect its fuzzing effectiveness. This conclusion has been confirmed in our study (to be presented in Section IV-C). Therefore, fuzzing MLIR compiler infrastructure is still an open challenge.

In this work, we proposed the first fuzzing technique to MLIR compiler infrastructure, called MLIRSmith. Instead of transforming high-level source programs targeting some specific compilers into MLIR programs for indirect fuzzing, MLIRSmith directly generates MLIR programs to fuzz MLIR compiler infrastructure. To ensure the fuzzing effectiveness, MLIRSmith considers both validity and diversity of generated MLIR programs by carefully exploiting the characteristics of MLIR compiler infrastructure. The validity of generated MLIR programs refers to conforming to the syntactic and semantic rules of MLIR, while the diversity refers to making generated MLIR programs cover as many combinations of operations in various dialects as possible.

To accomplish the goal, MLIRSmith generates MLIR programs in two phases. First, MLIRSmith performs MLIR program template construction. It tentatively masks the semantic details (i.e., attributes and operands) in each operation and focuses on covering different combinations of operations in various dialects in order to ensure program diversity. This phase is guided by a set of extended MLIR grammars in order to ensure syntactic validity. Second, MLIRSmith performs MLIR program instantiation, which aims to generate valid MLIR programs by instantiating a program template. Instantiating attributes and operands in an operation should conform to complex semantic rules (e.g., constraints between the operation and operands, different operands, or operands and attributes). To facilitate the instantiation process, we define a context-sensitive grammar to model these complex semantic rules, which allows MLIRSmith to properly instantiate an operand or an attribute according to its context, i.e., the settings of its dependent materials (operation, operands, or attributes) for ensuring semantic validity.

Such a two-phase generation strategy makes MLIR program generation more extensible, which opens the door for human-provided templates to incorporate expert knowledge in the process of fuzzing. For each generated MLIR program, MLIRSmith randomly selects passes in MLIR compiler infrastructure to transform or optimize it, and takes *crash* as the test oracle.

To evaluate MLIRSmith in fuzzing MLIR compiler infrastructure, we applied MLIRSmith to fuzz its latest reversion (fdbc55a). During two-month fuzzing, MLIRSmith detected 53 previously unknown bugs in total, where 49 of them have been confirmed and 38 have been fixed by developers. The bugs detected by MLIRSmith are diverse, which involve a wide range of root causes and passes in MLIR compiler infrastructure. We also compared MLIRSmith with NNSmith (the state-of-the-art high-level source program generator for deep learning compilers). We transformed the high-level source programs generated by NNSmith into MLIR programs with two frontends. Hence, we constructed two baselines: NNSmith (IREE) and NNSmith (ONNX-MLIR). During 24-hour fuzzing (by repeating five times), MLIRSmith detected 23 bugs while the two baselines detected only 6 and 7 bugs, respectively. Moreover, MLIRSmith covered 1.95× and 2.36× more lines, 2.26× and 2.78× more branches, 4.00× and 6.00× more dialects, 3.57× and 3.05× more operations than both baselines, respectively. The results demonstrate the significant superiority of MLIRSmith over indirect fuzzing through the high-level source program generator.

To sum up, our work makes the following contributions:

- We are the first to formulate and motivate the problem of fuzzing MLIR compiler infrastructure.
- We propose the first fuzzing technique for MLIR compiler infrastructure, MLIRSmith, by automatically generating valid and diverse MLIR programs.
- We conducted an extensive study to demonstrate the effectiveness of MLIRSmith. In particular, MLIRSmith detects 53 previously unknown bugs in the latest reversion of MLIR compiler infrastructure, where 49/38 have been confirmed/fixed by developers.

II. MLIR Compiler Infrastructure

Instead of designing and implementing a new IR with a single abstraction for each domain, MLIR compiler infrastructure provides a general infrastructure to support the development of various domain-specific compilers. Specifically, it provides systematic *passes* to support a wide range of functionalities for benefiting multiple domains simultaneously and introduces *dialects* to support multi-level IR.

Dialects are used to represent different levels of abstraction in MLIR. Each dialect defines a set of operations specific to a certain domain, such as linear algebra and machine learning. Specifically, dialects allow MLIR to support the representation of operations at different levels of abstraction (from low-level hardware-specific operations to high-level domain-specific abstractions), which actually forms an MLIR program. An MLIR program example is shown in Fig. 1b. It is helpful to optimize various operations at the proper level in order to achieve more efficient code generation and maximize the optimization ability for different hardware targets. By introducing or modifying dialects, an abstraction level can be conveniently added or refined to support new domains or hardware targets, while still maintaining consistent representation across different levels of abstraction. Hence, MLIR is more maintainable and flexible.

An **operation** represents a fundamental unit of computation in MLIR, which consists of its name, operands, and results (e.g., `memref.alloc`, `%rb`, and `%alloc` for the operation at Line 2 in Fig. 1b). The operands are the inputs required by the operation, each of which may have complex type constraints, and the results are its produced outputs. An operation could optionally have attributes and regions, where attributes are key-value pairs that provide additional information to specify the behavior of an operation (e.g., the alignment attribute for the `memref.alloc` operation at Line 2 is to specify the alignment of the allocated memory for the operation) and regions are used to represent nested blocks of operations (e.g., the `scf.parrallel` operation is nested in the function region and it also has an attached region from Line 3 to Line 6 that describes the parallel loop in Fig. 1b).

MLIR programs have complex semantic rules, which are a set of constraints that control program structure and behaviors. Note that in this work, we consider static semantics, while dynamic semantics (e.g. Undefined Behaviors) are not guaranteed. Specifically, static semantics control the use of operand types, attributes, and regions, as well as their interactions. We explain each type of rules in detail as follows:

- **Semantic rules on attributes**: This type of rules ensure that (1) the attributes are consistent with the semantics of the operation and (2) they do not conflict with other attributes. For instance, the alignment attribute of the `memref.alloc` operation (at Line 2 in Fig. 1b) must be a non-negative integer.
- **Semantic rules on operand types**: The operands of an operation must be of compatible types. That is, the types of operands must be (1) consistent with the semantics of the operation and (2) compatible with each other. For instance, the `memref.store` operation (at Line 4 in Fig. 1b) requires the type of the second operand to be `memref` type and the element type (i.e., `f32`) of `memref` must be the same as the type of the first operand.
- **Semantic rules on operand visibility**: The operations can only use SSA (i.e., Single Static Assignment [19]) values that are in scope. A visible value is the one that is defined within the same region or a parent region of the operation. One exceptional case is the `func.func` operation, which prevents the operations inside the region from using SSA values defined outside. For instance, the SSA value `%alloc` is visible to the `memref.store` operation (at Line 4 in Fig. 1b) .
- **Semantic rules on regions**: Some operations can only appear in some specific regions, and in turn, the regions of some operations can only contain some specific operations. For instance, the terminator (that marks the end of a region) of a function must be the `return` operation and the `return` operation can only serve as the terminator.

Passes implement various transformations or optimizations on MLIR programs, especially operations under various dialects. MLIR compiler infrastructure organizes most of its functionalities into passes. Some passes are designed to perform common transformations or optimizations to multiple dialects, such as common subexpression elimination, while some passes are designed for a specific dialect, such as the "affine-loop-unroll" pass that unrolls loops in the `affine` dialect in order to improve performance. It is also convenient to define new customized passes to meet some specific demands in the MLIR compiler infrastructure.

III. MLIRSmith

A. Design of MLIRSmith

In this work, we propose the first fuzzing technique, called MLIRSmith, for the MLIR compiler infrastructure. The core of MLIRSmith is to generate valid and diverse MLIR programs in order to achieve effective and efficient fuzzing. Regarding the validity of MLIR programs, MLIRSmith requires to ensure both syntactic validity and semantic validity, which is helpful to evaluate deep functionalities in MLIR compiler infrastructure. This is because a syntactically invalid MLIR program can be directly rejected by the parser, while a semantically invalid MLIR program can be detected at the very early stage of MLIR passes. Generating diverse MLIR programs is helpful to improve the fuzzing effectiveness by covering more MLIR compiler infrastructure code and exploring input space more sufficiently. The core of MLIR programs lies in operations and dialects, and thus different combinations of operations in various dialects can help enhance the diversity of generated MLIR programs.

To accomplish the above goal, MLIRSmith employs a two-phase program generation process, i.e., constructing MLIR program templates and generating MLIR programs by instantiating program templates. The former pays attention to improving the diversity of generated MLIR programs by incorporating diverse operations and dialects, and tentatively ignores semantic details (i.e., operands and attributes) in each operation. In particular, the construction of program templates is guided by the extended MLIR syntax in order to ensure the syntactic validity of generated MLIR programs. The latter aims to instantiate program templates to generate valid MLIR programs. The instantiation process carefully considers complex semantic rules through designing a context-sensitive grammar to ensure the semantic validity of generated MLIR programs. The two-phase strategy makes MLIR program generation more extensible. It is helpful to incorporate expert knowledge by utilizing human-provided MLIR program templates to generate MLIR programs for fuzzing.

In the following, we first introduce the phase of MLIR program template construction in Section III-A, then present the phase of MLIR program instantiation in Section III-C, and finally illustrate the overall fuzzing process with MLIRSmith in Section III-D.

B. MLIR Program Template Construction

We design syntax rules for MLIR program templates, which extend a subset of MLIR grammars and are shown in Fig. 2. Note that the `?` in the second production rule represents a dynamic dimension. An MLIR program template consists of

```
1. func.func @parallel_store(%cst: f32,  %lb: index, %rb: index,
        %step: index) {
2.    %alloc = memref.alloc [[V1]] {alignment = [[C1]]} : memref<?xf32>
3.    scf.parallel (%iv) = [[V2]] to [[V3]] step [[V4]] {
4.       memref.store [[V5]], [[V6]] [[[V7]]]
5.       scf.yield
6.    }
7.    return
8. }
```

(a) Program template with placeholders

```
1. func.func @parallel_store(%cst: f32,  %lb: index, %rb: index,
        %step: index) {
2.    %alloc = memref.alloc(%rb) {alignment = 8} :  memref<?xf32>
3.    scf.parallel (%iv) = (%lb) to (%rb) step (%step) {
4.       memref.store %cst, %alloc[%iv] : memref<?xf32>
5.       scf.yield
6.    }
7.    return
8. }
```

(b) Program generated by instantiating the template

Fig. 1: Motivating example

$$
\begin{aligned}
Id &::= dialect.op \\
Dim &::= \mathbb{N}^{+} \mid ? \\
Shape &::= \epsilon \mid Dim \mid Dim{\times}Shape \\
Type &::= \texttt{i1} \mid \texttt{i32} \mid \texttt{i64} \mid \texttt{f32} \mid \texttt{f64} \mid \texttt{index} \\
&\quad \mid \texttt{tensor<}Shape{\times}Type\texttt{>} \\
&\quad \mid \texttt{memref<}Shape{\times}Type\texttt{>} \\
&\quad \mid \texttt{vector<}Shape{\times}Type\texttt{>} \\
Operand &::= [[V]] \\
Attr &::= key : [[C]] \\
Operation &::= Id\ (\{Attr^{*}\})?\ Operand^{*}\ Region^{*} \\
Region &::= \{Type^{*} Operation^{*}\} \\
Program &::= \\
&(\texttt{func.func}\ Attr^{*}\ Type^{*}{\rightarrow}Type^{*}\ Region?)^{*}
\end{aligned}
$$

Fig. 2: The syntax rules for MLIR program templates.

multiple functions, where a function consists of the operation identifier (i.e., `func.func`), the function signature (the function name specified by attribute and a list of argument types), the return types, and the function body that is a region consisting of a list of operations. The operation is the basic unit for our program template generation and contains an *Id*, an optional list of attributes, operands (i.e., values represented in %SSA symbols), and an optional region. We introduce the placeholders $[[V]]$ and $[[C]]$ to mask the semantic details of an operation and focus on diversifying the combinations of dialects and operations in a template in this phase. $[[V]]$ represents available values for an operand, and $[[C]]$ represents available literals or constants for an attribute. Therefore, the program templates can express a set of MLIR programs by instantiating the placeholders.

To generate a program template, we need to first decide the number of functions within the program using a hyper-parameter. For each function, the function signature (which consists of a list of attributes and types) and the return value (which consists of a list of types), are randomly generated. The function body, which is a region, is generated by Algorithm 1. The algorithm takes *scope* that defines operations available to the region, and the configuration that allows users to set the maximum depth of nested regions and the maximum number of operations within a region, as input, and output the generated region template.

Algorithm 1 starts by generating an empty region template *tmpl* at Line 2. Then, the operations available for sampling in the generation process is obtained through *getAvailableOp* (at Line 3). This step filters out the operations that do not satisfy the depth limit constraint by the configuration from *scope*. The loop (from Line 4 to 11) fills *tmpl* with a list of operation templates. Each iteration first adaptively samples an operation Id *opId* from *availOpIds* (at Line 5). An operation has a greater probability if it is less sampled in history. The operation template with *opId* is created (at Line 6) by masking the operand and the attribute of the operation. Then, the operation template is appended to the operation list of the region template *tmpl.ops* (at Line 7). If the generated operation has a region, the region is recursively generated (at Line 9) according to *conf* (to avoid exceeding the region depth limit). Then, the abstract terminator operation is created and appends to *tmpl.ops* (Line 12 and 13).

We use Fig. 1a to illustrate how MLIRSmith generates the example template with Algorithm 1. MLIRSmith starts with generating functions using the `func.func` operation. MLIRSmith generates the function name `@parallel_store` and a list of types `f32, index, index, index` as its signature, an empty list as its result types, and its function body region. Specifically, the region is generated through Algorithm 1. It generates the operation template sequence [`memref.alloc`, `scf.parallel`] as well as the terminator `return` for the function body region, where the operands and attributes of each operation are masked with placeholders, for instance, the `alignment` attribute and the only operand of `memref.alloc` is masked as $[[C_1]]$ and $[[V_1]]$). Algorithm 1 is recursively applied to generate a region for operation `scf.parallel` (at Line 3 in Fig. 1a).

C. MLIR Program Instantiation

Given a generated program template, MLIRSmith designs a context-aware program instantiation strategy, which enforces semantic rules in the generated MLIR programs. We first define a context-sensitive grammar inspired by the existing

Algorithm 1: Generating Region Template

```
    Input:  scope: defining operations available to the region
            conf: configuration for generation
    Output: tmpl: generated template
1  Function RegionTmplGeneration(scope, conf):
2      tmpl := initialize an empty region;
3      availOpIds :=
         getAvailableOp(scope, conf.depthLimit);
4      while len(tmpl.ops) ≤ conf.lenLimit do
5          opId := adaptive sampling from availOpIds;
6          op := operation template with opId;
7          tmpl.ops.append(op);
8          if op has a region then
9              child := RegionTmplGeneration(opId, conf);
10         end
11     end
12     t := generate abstract terminator;
13     tmpl.ops.append(t);
14     return tmpl;
```

Algorithm 2: Instantiating templates

```
    Input:  op: operation with placeholder
    Output: op: operation instance
1  Function TemplateInstantiation():
2      vals := set of visible values at the current program point;
3      types := list of known types;
4      foreach key in op.attrs do
5          op.attrs[key] := generateAttr(op.id, key);
6      end
7      foreach operand in op.operands do
8          type := generateType(op.id, op.attrs, types);
9          types.append(type);
10         candidates := generateOperand(vals, type);
11         if candidates is an empty set then
12             newVal := init value with type;
13             candidates.add(newVal);
14             vals.add(newVal);
15         end
16         operand := sample from candidates;
17     end
18     return op
```

work [20], based on which we define context-sensitive production rules to express the semantics.

Definition 1 (Context-sensitive grammar (CSG)): A context-sensitive grammar is a 4-tuple $G = (N \cup H, \sum, R, s)$, where

- N is a finite set of non-terminal symbols and H is the set of placeholder symbols,
- $\sum$ is a finite set of terminal symbols,
- R is a finite set of context-sensitive production rules with the form of $[c]\alpha \rightarrow \beta_1\beta_2 \dots \beta_n$, where $\alpha \in N \cup H$, $n \geq 1$, $\beta_i \in (N \cup \sum)$ for $i = \{1, \dots, n\}$, and c is the context in the form of $<S_v, ID, A, \mathcal{T}>$, in which S_v is the visible value set, ID is the operation id, A is the attribute set and $\mathcal{T}$ is the list of operand types.
- $s \in N$ is an unique start symbol.

We design a set of context-sensitive production rules, which take into account the semantic rules for the operands and attributes of an operation, based on the context-sensitive grammar.

***Rule Class** 1:* $[<\Phi, ID, \Phi, []>]\ [[C_{key}]] \rightarrow c$
Rule class 1 contains rules that instantiate the attribute placeholder $[[C_{key}]]$ of the corresponding key based on the operation *ID*, where the mapping from the operation *ID* to the set of attribute key is adopted from official documents [21]. There are a total of 202 such mapping pairs and thus 202 concrete rules in this class. The first example in Table I provides a concrete rule of this class, the `memref.alloc` operation requires the value of the key `alignment` to be a non-negative integer.

***Rule Class** 2:* $[<S_v, ID, A, [T_1, \dots, T_{i-1}]>]\ [[V_i]] \rightarrow Value$
Rule class 2 instantiates the value of the ith operand placeholder based on the visible value set (S_v), operation *ID*, the attributes (A), and the types of its preceding $i-1$ operands, i.e., $T_1, \dots, T_{i-1}$. For instance, the second example in Table I shows a production rule that instantiates the placeholder $[[V_6]]$ with the value `%alloc`, since the `memref.store` operation stores the first operand (i.e., value to be stored) into the second operand (i.e., the memory to be operated), the type of the second operand must be `memref` and its element type should be consistent with the type of the first operand. Then, the $[[V_6]]$ is replaced with the value of the type `memref<?×f32>` in S_v (i.e., `%alloc`) to satisfy the requirement.

Algorithm 2 illustrates the instantiation process from a program template based on the context-sensitive production rules. The algorithm takes as input an operation with placeholders *op* and the visible value set *vals*.

First, it applies Rule Class 1 to generate literals or constants that satisfy the constraints for each key of the operation attribute (Lines 4-6). Then, it proceeds to fill the operands of the operation with visible values while adhering to the type constraints specified in Rule Class 2. The algorithm maintains the former $i-1$ determined operand types (with variable $types$ at Line 3) and iterates over the operands in operation (Lines 7-17). For each operand, it first infers a proper type and updates the *types* list accordingly (Lines 8-9). It then searches the visible value set to obtain value candidates that match the operand type (Line 10). If no proper operands can be found, the algorithm generates an initial value of the operand type and adds it to the visible value set (Lines 11-15). Finally, the algorithm selects an operand from the candidate set and inserts it into the operand list of the operation (Line 16). Note that the functions `generateAttr` and `generateType` randomly select an attribute and a type from allowable ones, respectively.

Based on the above algorithm, MLIRSmith instantiates the program template shown in Fig. 1a to produce an MLIR program as shown in Fig. 1b. The placeholders in the program template are instantiated by assigning concrete values to operands and concrete literals to attributes. For example, at Line 2, the placeholder $[[C_1]]$ is instantiated by the integer 8, and $[[V_1]]$ is instantiated by the `index` value `%rb`.

TABLE I: Examples of context-sensitive production rules

ID	Production Rule Example
1	[<Φ, memref.alloc, Φ, Φ>] [[$C_{alignment}$]] $\rightarrow \mathbb{N}$
2	[<{%cst: f32, %alloc: memref<?×f32>, ...}, memref.store, {}, [f32]>] [[V_6]] $\rightarrow$ %alloc

? in memref<?×f32> indicates that the first dimension of the memory is dynamic

Algorithm 3: Fuzzing with MLIRSmith

```
   Input: opts: initial options
   Output: cps: crashed tests and the corresponding option
 1 Function MLIRCompilerFuzzing(opts):
 2     while time not exceed do
 3         test := new program generated by invoking
            RegionTmplGeneration and
            TemplateInstantiation;
 4         sOpts := empty set of pass options;
 5         foreach opt in opts do
 6             stat := state of executing test with option opt;
 7             if stat indicates a crash then
 8                 cps.add(test, opt);
 9             end
10             else
11                 sOpts.add(opt);
12             end
13         end
14         while not exceed the trials limit do
15             optSeq := random select a sequence from
                sOpts;
16             stat := state of executing test with option
                optSeq;
17             if stat indicates a crash then
18                 cps.add(test, optSeq);
19             end
20         end
21     end
22     return cps
```

D. Overall Fuzzing Process with MLIRSmith

We applied MLIRSmith to fuzz the MLIR compiler infrastructure by randomly generating MLIR programs. To make an MLIR program execute the MLIR compiler infrastructure, we randomly enable a sequence of passes to transform and optimize the MLIR program. If a crash occurs, we regard the MLIR program triggers an MLIR compiler infrastructure bug.

The fuzzing process is shown in Algorithm 3, which takes the whole set of passes *opts* as input. The fuzzing process is iterative and terminates until the pre-defined time budget is exhausted. At each iteration, MLIRSmith applies each individual pass and a pre-defined number of pass sequences to transform/optimize each MLIR program (denoted as *test* at Line 3). MLIRSmith first applies each *opt* in *opts* to *test*, and observes the execution status *stat* of *test* (Lines 5-13). If the *stat* indicates a crash, a bug is detected (at Line 7); otherwise, the *opt* is added to *sOpts*. Since the passes that can trigger crashes individually are not included in $sOpts$, we can reduce the possibility of triggering duplicate crashes by constructing pass sequences from $sOpts$. Specifically, we then use *test* for fuzzing under a pre-defined number of pass sequences (Lines 14-20). Each pass sequence is constructed by random sampling passes from $sOpts$ at Line 15. After applying the pass sequence to *test* (Line 16), we observe whether a crash occurs (Lines 17-18).

IV. Evaluation

To evaluate MLIRSmith, we aim to address the following research questions (RQs):

- **RQ1**: Can MLIRSmith detect previously unknown bugs in MLIR compiler infrastructure?
- **RQ2**: Can MLIRSmith outperform indirect fuzzing techniques (transforming high-level source programs to MLIR programs for fuzzing) in terms of test effectiveness?

A. Experimental Setup

We evaluated MLIRSmith on the recent revision (fdbc55a) of MLIR compiler infrastructure in order to detect previously unknown bugs. It has more than 392K lines of code.

Baselines: Since MLIRSmith is the first fuzzing technique specific to MLIR compiler infrastructure, we did not have direct baselines. In fact, some high-level source programs can be transformed into MLIR programs and thus can also be adapted to fuzz MLIR compiler infrastructure. As explained aforementioned, such methods are indirect and do not consider the characteristics of MLIR compiler infrastructure, which can limit the test effectiveness. To sufficiently evaluate MLIRSmith, we still migrated such an indirect technique to fuzz MLIR compiler infrastructure for comparison by adopting NNSmith [14] as the representative. Specifically, NNSmith is the state-of-the-art test generator for fuzzing deep learning compilers, which can automatically generate ONNX computation graphs. To use its generated tests for fuzzing MLIR compiler infrastructure, we transformed ONNX computation graphs generated by NNSmith into MLIR programs through corresponding frontends. In particular, there are two available frontends supporting such transformations, i.e., IREE [10] and ONNX-MLIR [4]. In our study, we use both of them for sufficient comparison and call the baselines **NNSmith (IREE)** and **NNSmith (ONNX-MLIR)**, respectively. Here, we adopted the default configurations of NNSmith provided by the existing work [14] for evaluation.

Metrics: In the study, we used three metrics to measure the effectiveness: the number of detected bugs, the number of covered lines and branches in MLIR compiler infrastructure, and the number of covered dialects and operations. The first one is the main metric in our study as well as many

existing studies on fuzzing [14], [17], [12], [22], [23], [24]. During the fuzzing process, each technique produced a set of crashes triggered by MLIR programs. We de-duplicated them according to crash messages and submitted unique crashes to MLIR compiler infrastructure developers. Based on their feedback, we counted the number of detected bugs. The second one is also important in the area of fuzzing and has been widely used in the existing work [12], [25], [13]. The third one is to measure the diversity of generated MLIR programs. We considered the number of covered dialects (and operations), the number of covered dialect pairs (and operation pairs) with control dependency, and the number of covered dialect pairs (and operation pairs) with data dependency. The latter two reflect the combination of dialects (and operations) to some degree.

Configurations: The *conf* in Algorithm 1 is described as follows: the maximum region depth is 3 and the maximum number of operations in a region is 128. Each template contains exactly 1 function. For Algorithm 3, the input *opts* is collected from the official pass document [26].

To sufficiently evaluate MLIRSmith, we ran it for 2 months to detect as many previously unknown bugs as possible in order to help enhance the quality of MLIR compiler infrastructure. To compare with baselines, we ran each technique for 24 hours. To reduce the influence of randomness, we repeated the comparison experiments 5 times with different random seeds and reported the aggregated results for each technique.

Implementation: We implemented MLIRSmith on top of the MLIR compiler infrastructure. It has 9,923 lines of C++ code. The current version of MLIRSmith supports 12 widely-used dialects (i.e., `builtin`, `func`, `arith`, `tensor`, `memref`, `linalg`, `vector`, `index`, `math`, `scf`, `affine`, `bufferization`) and all the operations in these dialects. Due to the implementation effort, we leave the remaining dialects as future work. The remaining dialects tend to be more low-level, such as hardware-specific dialects[27]. Intuitively, the test capability of MLIRSmith can be further improved by incorporating the remaining dialects.

Data Availability: We released our tool MLIRSmith and experimental data at our project homepage for experiment replication and practical use [28]. We hope that the artifact can be helpful to promote future research in this field.

Environment: Our study was conducted on a machine with Intel(R) Xeon(R) Gold 6240C CPU @ 2.60GHz and 128G Memory, Ubuntu 18.04.6 LTS.

B. RQ1: Previously Unknown Bugs Detected by MLIRSmith

During two-month fuzzing, MLIRSmith detected 53 previously unknown bugs in total. The details of these bugs are presented in Table II, where each column represents the bug ID, the root cause of the fixed bug, the type of the bug-occurring pass, and the bug status labeled by developers. Among the 53 bugs, 49 have been confirmed and 38 have already been fixed by MLIR compiler infrastructure developers. The remaining four are awaiting feedback from the developers.

The bugs detected by MLIRSmith are diverse. They involved a wide range of (1) passes in MLIR compiler infrastructure and (2) root causes. Then, we analyzed these bugs from the two aspects.

Pass Analysis: According to the document of MLIR compiler infrastructure, the passes are categorized as bufferization passes, conversion passes, general transformation passes, dialect-specific passes, and others.

- *Bufferization passes* are responsible to convert operations with tensor semantics to operations with memref semantics by replacing the tensor operands with memref values. Among the 53 reported bugs, only 2 occur in the bufferization passes.
- *Conversion passes* perform transformations between dialects to lower the abstraction level. For example, the pass "convert-scf-to-cf" transforms structured control flow (which can be organized in nested structure) to flattened control flow (that only has branches or conditional branches). There are a total of 19 bugs that have been identified to occur in the conversion passes.
- *General transformation passes* can be applied to all dialects and are responsible to perform common optimizations/transformations, such as common subexpression elimination. Specifically, 9 bugs occur in general transformation passes, including 5 bugs in *canonicalization*, 1 bug in *common subexpression elimination*, and 3 bugs in *inline*.
- *Dialect-specific passes* are responsible to perform optimizations/transformations within each specific dialect. For example, the pass "affine-loop-unroll" is specific to optimizing the operations within the "affine" dialect. There are 19 bugs occur in dialect-specific passes, including 5 in passes for the "affine" dialect, 2 in passes for the "arith" dialect, 6 in passes for the "linalg" dialect, 2 in passes for the "llvm" dialect, 2 in passes for the "scf" dialect, and 2 in passes for the "sparse_tensor" dialect.
- There are also 4 bugs that do not belong to any of the four categories of passes, and we denoted them as "others".

Root Cause Analysis By analyzing the 38 fixed bugs according to developers' discussion and corresponding patches, we further summarized the root causes of these bugs. Moreover, for each root cause, we selected one bug as the illustrative example.

Incomplete Verifier (IV): Each pass has a verifier to check the compatibility between the pass and some operations. This root cause is that a necessary verifier of a pass is missing or incomplete, which causes the pass works on incompatible operations, leading to a crash. 7 (out of 38) fixed bugs are due to this root cause. For example, as shown in Fig. 3, Bug #59496 was caused by the "convert-tensor-to-spirv" pass working on the `arith.extsi` operation with an incompatible type (i.e., `i64`). Hence, a crash occurred. Subsequently, it is fixed by adding a verifier to check such cases in order to ensure compatibility between the pass and the operation.

Incorrect Pattern (IP): Each pass uses a set of patterns to

TABLE II: Details of Submitted/Confirmed/Fixed bugs detected by MLIRSmith

Bug Id	Root Cause	Pass Category	Status	Bug Id	Root Cause	Pass Category	Status
#58258	-	Dialect-Specific (affine)	confirmed	#59714	-	Conversion	confirmed
#58411	IRL	Dialect-Specific (linalg)	fixed	#59970	Others	Dialect-Specific (sparse_tensor)	fixed
#58745	IRL	General Transformation	fixed	#59972	-	Others	confirmed
#58746	IRL	General Transformation	fixed	#59986	IV	Conversion	fixed
#58747	IP	Dialect-Specific (linalg)	fixed	#59987	IA	Conversion	fixed
#58748	-	Dialect-Specific (linalg)	confirmed	#59989	IRL	Conversion	fixed
#58749	IP	Conversion	fixed	#59991	-	Dialect-Specific (linalg)	confirmed
#58803	-	Conversion	confirmed	#59993	IA	Conversion	fixed
#58805	UD	Conversion	fixed	#59994	IRL	Dialect-Specific (affine)	fixed
#58807	IP	Dialect-Specific (arith)	fixed	#60018	-	Conversion	confirmed
#58869	-	Dialect-Specific (linalg)	confirmed	#60070	UD	Conversion	fixed
#59135	IRL	General Transformation	fixed	#60093	IP	General Transformation	fixed
#59136	IP	Conversion	fixed	#60094	IRL	Dialect-Specific (scf)	fixed
#59182	IV	Conversion	fixed	#60186	Others	Others	fixed
#59234	IRL	Dialect-Specific (affine)	fixed	#60193	IRL	General Transformation	fixed
#59442	-	Bufferization	confirmed	#60195	IV	Dialect-Specific (sparse_tensor)	fixed
#59443	IRL	Dialect-Specific (scf)	fixed	#60197	UD	Conversion	fixed
#59444	IA	Dialect-Specific (affine)	fixed	#60199	IV	Conversion	fixed
#59445	-	Bufferization	submitted	#60214	IP	Others	fixed
#59454	-	Conversion	confirmed	#60216	-	Others	confirmed
#59455	IRL	General Transformation	fixed	#59725	-	Dialect-Specific (linalg)	submitted
#59460	IRL	Dialect-Specific (llvm)	fixed	#59726	-	General Transformation	submitted
#59461	IRL	Dialect-Specific (affine)	fixed	#61054	IA	Conversion	fixed
#59462	UD	Dialect-Specific (llvm)	fixed	#61056	IV	General Transformation	fixed
#59496	IV	Conversion	fixed	#61094	IP	Conversion	fixed
#59617	IP	Dialect-Specific (arith)	fixed	#61380	IRL	Conversion	fixed
#59703	-	General Transformation	submitted				

```
1. func.func @type_conversion_failure(%arg0: i32) {
2.   %1 = arith.extsi %arg0 : i32 to i64
3.   return
4. }
```

Fig. 3: Bug#59496: Incomplete Verifier

```
1. %res = scf.if %cond -> (memref<8xi32>) {
2.   scf.yield %arg0 : memref<8xi32>
3. } else {
4.   scf.yield %arg1 : memref<8xi32>
5. }
```

Fig. 4: Bug#59136: Incorrect Pattern

match the operations that it will transform/optimize. If some patterns are incorrect, a pass will transform/optimize unexpected operations, leading to a crash. 9 fixed bugs are caused by this root cause. For example, Bug #59136 was triggered by the MLIR program shown in Fig. 4. This program contains a `scf.if` operation that returns a memref without specifying SPIR-V Storage Class, which should not be processed by the "convert-scf-to-spirv" pass. Due to incorrect pattern, the pass unexpectedly operates on this operation. This bug was fixed by correcting the pattern to avoid mismatching this kind of operations.

```
1. vector.reduction <add>, %arg0 : vector<f32> into f32
```

Fig. 5: Bug#60193: Incorrect Rewrite Logic

```
1. %0 = vector.transfer_read %arg[], %f0 { ... } :
     tensor<f32>, vector<1xf32>
```

Fig. 6: Bug#60197: Unregistered Dialect

Incorrect Rewrite Logic (IRL): During the transformation/optimization process, passes rewrite matched operations to new ones. However, if the rewrite logic is incorrect, the pass will produce incorrect operations. 13 fixed bugs belong to this root cause. For example, Bug #60193 was triggered by the program shown in Fig. 5, which contains a `vector.reduction` operation that summed up all the elements in the vector `%arg0`. The canonicalization pass aims to rewrite it to a `vector.extract` operation, but ignores the case where the rank is 0. That is, the logic of this pass just considers the cases where the rank is at least 1, leading to a crash on this program. By modifying the rewrite logic to include this case, this bug was fixed.

Unregistered Dialect (UD): To accomplish the transformation between an operation in a dialect and an operation in another dialect, the latter dialect should be registered in the pass. Otherwise, the transformation will crash and fail to

```
1. #map = affine_map<()[s0] -> (s0)>
2. func.func @func(%arg0: memref<?xf32>, %0: index){
3.   %dim = memref.dim %arg, %0 : memref<?xf32>
4.   %1 = affine.apply #map()[%dim]
5.   return
6. }
```

Fig. 7: Bug#59993: Incorrect Assertion

create the target operation. 5 fixed bugs are caused by this root cause. For example, Bug #60197 was triggered by the program shown in Fig. 6 under the "convert-vector-to-scf" pass. This pass attempted to transform the `vector.transfer_read` operation to a `tensor.extract` operation, but since the `tensor` dialect was not registered in the pass, causing that the transformation cannot be completed. To resolve this bug, the missing dialect has to be registered in the pass.

Incorrect Assertion (IA): There are a lot of assertions in MLIR compiler infrastructure, which are used to verify internal states by specifying invariants at certain code locations. If an assertion is incorrect, the transformation/optimization process will crash even though the internal state is correct. That is, it will prevent valid programs from being processed. 4 fixed bugs belong to this root cause. For example, Bug#59993 was triggered by the program shown in Fig. 7. At Line 4, the `#map` attribute of the `affine.apply` operation takes `%dim` as a symbol. This is a valid program that should be appropriately transformed. However, the assertion in the "scf-for-loop-peeling" pass incorrectly assumes that `%dim` must have a concrete value, although it is retrieved from a memref whose dimension size cannot be statically determined (Line 3). This leads to a crash. Fixing the assertion resolved this bug.

C. RQ2: Comparison between MLIRSmith and NNSmith

For sufficient evaluation, we compared MLIRSmith with two baselines, i.e., NNSmith (IREE) and NNSmith (ONNX-MLIR). NNSmith is a high-level source program generator for deep learning compilers, which is indirect to fuzz MLIR compiler infrastructure by transforming high-level computation graphs to MLIR programs through two frontends. We ran each technique for 24 hours and repeated the experiment five times to show the aggregated results.

Fig. 8a shows the number of bugs detected by each technique and analyzes their overlap. From this figure, MLIRSmith detected 23 bugs, while NNSmith (IREE) and NNSmith (ONNX-MLIR) detected only 6 and 7 bugs, respectively. In particular, 15 out of 23 bugs detected by MLIRSmith cannot be detected by both baselines. The results demonstrate the significant superiority of developing an effective MLIR program generator over the baseline techniques indirect to fuzz MLIR compiler infrastructure. Although both baselines detected a few unique bugs in this experiment, all of them can be detected by MLIRSmith during the two-month fuzzing. However, these unique bugs detected by MLIRSmith cannot be detected by both baselines with longer fuzzing time due to the limited diversity of their generated MLIR programs, which will be discussed in detail later.

TABLE III: Diversity comparison

Metric	MLIRSmith	NNSmith (IREE)	NNSmith (ONNX-MLIR)
dialects	**12**	3	2
dialect pairs (control/data)	**62 / 89**	3 / 2	2 / 1
operation	**125**	35	41
operation pairs (control/data)	**1772 / 2199**	39 / 338	47 / 153

Further, we investigated the reason for the superiority of MLIRSmith by measuring the covered lines and branches in MLIR compiler infrastructure and the diversity of generated MLIR programs. Fig. 8b and Fig. 8c show the number of lines and branches covered by MLIRSmith and baselines as well as their overlap analysis results, respectively. From these figures, MLIRSmith covered 1.95× and 2.36× more lines and 2.26× and 2.78× more branches than NNSmith (IREE) and NNSmith (ONNX-MLIR), respectively. In particular, 24,918 lines and 19,539 branches covered by MLIRSmith cannot be covered by both baselines.

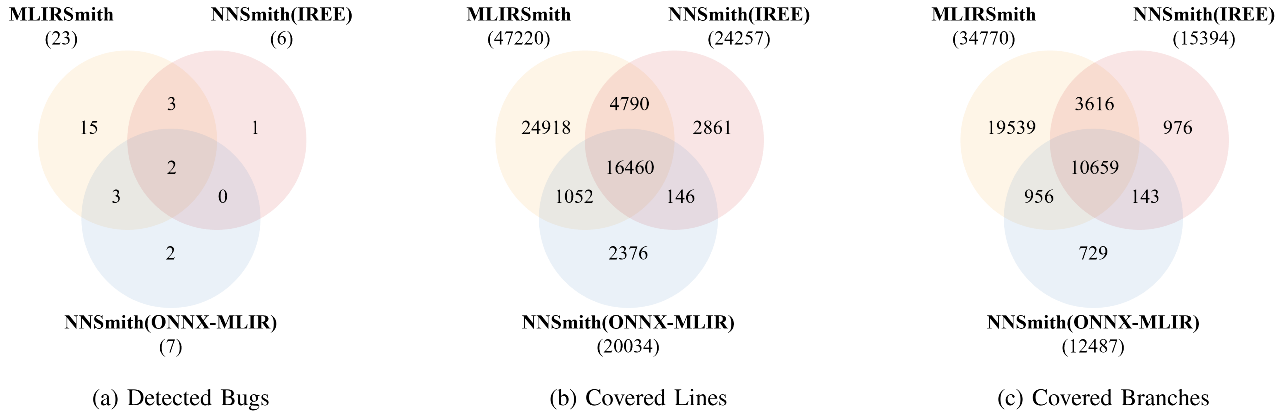
We then compared MLIRSmith and baselines in terms of the diversity of their generated MLIR programs. Table III shows the number of covered dialects (and operations), the number of covered dialect pairs (and operation pairs) with control dependency, and the number of covered dialect pairs (and operation pairs) with data dependency, respectively. From this table, MLIRSmith outperforms both baselines in terms of all these metrics. Specifically, the generated MLIR programs by MLIRSmith covered 4.00× and 6.00× more dialects, 20.67× and 31.00× more dialect pairs with control dependency, 44.50×, and 89.00× more dialects pairs with data dependency, 3.57× and 3.05× more operations, 45.44× and 37.70× more operation pairs with control dependency, and 6.51× and 14.37× more operation pairs with data dependency than NNSmith (IREE) and NNSmith (ONNX-MLIR), respectively.

Overall, the diversity of the MLIR programs transformed from high-level source programs generated by NNSmith is limited, which tends to explore fixed patterns of dialects and operations and thus leads to less code coverage and fewer detected bugs than MLIRSmith. Specifically, NNSmith (IREE) can only cover `tosa`, `func` and `builtin` dialects, and NNSmith (ONNX-MLIR) can only cover `llvm` and `builtin` dialects. This also confirms the contribution of developing an MLIR program generator (i.e., MLIRSmith), which can more flexibly explore dialects, operations, and their combinations, and thus significantly improve the fuzzing effectiveness.

D. Threats to Validity

The threat to *internal* validity mainly lies in the implementation of MLIRSmith. To reduce this threat, two authors carefully checked all the code and wrote test cases to test MLIRSmith.

The threat to *external* validity mainly lies in the subject used in our study. Indeed, there are many revisions of the MLIR



(a) Detected Bugs (b) Covered Lines (c) Covered Branches

Fig. 8: Comparison of the number of detected bugs, covered lines, and covered branches

compiler infrastructure, but we just used the latest revision as the subject. This is because it is more significant to detect previously unknown bugs. In the future, we can continue fuzzing future revisions with MLIRSmith.

The threats to *construct* validity mainly lie in randomness and baselines. To reduce the former threat, we repeated the comparison experiment with baselines for five times and reported the aggregated results. To reduce the latter threat, we tried our best to migrate NNSmith to fuzz the MLIR compiler infrastructure. Our study showed the limitation of such a high-level source program generator in MLIR compiler infrastructure fuzzing. In the future, we can try other high-level source program generators (as well as AFL [29] despite likely generating invalid programs) for more sufficient investigation.

V. Discussion

A. Efficiency

We investigated the efficiency of MLIRSmith. During 24-hour fuzzing, MLIRSmith ran 270 MLIR programs on average, while NNSmith (IREE) and NNSmith (ONNX-MLIR) ran 392 and 615 MLIR programs, respectively. Although MLIRSmith is less efficient than both baselines, it is much more effective, demonstrating its high cost-effectiveness. We further analyzed the reason why MLIRSmith is less efficient than both baselines. This is because the size of the MLIR programs generated by MLIRSmith is larger than those of both baselines, that is, the MLIR programs generated by MLIRSmith contain more operations. Transforming or optimizing them under the same pass tends to be more time-consuming.

B. Future Work

MLIRSmith can be further improved from the following aspects. First, we can make MLIRSmith support more dialects, which can directly improve its test effectiveness. Second, the current MLIRSmith mainly adopts a random process to generate MLIR programs. Many existing studies on fuzzing have demonstrated some guidance is helpful in improving the fuzzing effectiveness. In the future, we can incorporate effective guidance (such as coverage) to MLIRSmith. Third, the current MLIRSmith treats all the dialects equally. In the future, we can prioritize dialects to test more important functionalities in MLIR earlier. Fourth, due to the two-phase generation strategy in MLIRSmith, we can conveniently incorporate more effective templates for generating MLIR programs, which is also a promising direction. Lastly, the test oracle used in MLIRSmith is just *crash*. In the future, we can incorporate more test oracles to MLIRSmith, such as EMI-based test oracles [25], [30], [31].

C. Significance of Fuzzing MLIR Compiler Infrastructure

MLIRSmith is specific to the MLIR compiler infrastructure, but its significance is not limited to the single system. This is because many compilers are built on top of MLIR compiler infrastructure, fuzzing MLIR compiler infrastructure can help ensure the reliability of all of its powered compilers.

VI. Related Works

Our work is closely related to test program generation in compiler testing. The existing test program generation techniques are mainly divided into two categories: test program generation from scratch [32], [33], [12], [13], [14], [34] and test program generation from seed programs [35], [36], [17], [37], [25], [38], [29]. Our work belongs to the first category. In the following, we introduce related work from the two aspects. More details about compiler testing can be found in the survey [39].

Test program generation from scratch: This category of techniques generate test programs based on grammars to fuzz compiler testing. For example, Yang et al. [12] proposed Csmith, which randomly generates C programs with different code features. Based on Csmith, Lidbury et al. [13] proposed CLsmith, which lifts Csmith to generate OpenCL programs with six modes randomly. Liu et al. [14] proposed NNSmith, which generates ONNX computation graph programs for testing deep learning compilers. Ma et al. [34] proposed HirGen, which generates computation graph programs represented by Relay IR for testing TVM.

Different from them, our work proposes MLIRSmith to generate MLIR programs for fuzzing MLIR compiler infrastructure rather than the compilers specific to a domain. Moreover, MLIR programs have different characteristics (e.g., data structure and semantics) from the programs targeted by these existing techniques. In our study, we also compared MLIRSmith with the representative high-level source program generator (i.e., NNSmith), demonstrating the superiority of MLIRSmith in fuzzing MLIR compiler infrastructure.

Test program generation from seed programs: This technique category generates test programs by mutating or synthesizing test programs based on multiple seed programs. For example, Holler et al. [36] proposed LangFuzz to synthesize test programs by code snippet extraction and replacement based on seed programs for JavaScript engine fuzzing. Zhao et al. [17] proposed JavaTailor to generate test programs by inserting code ingredients from historically bug-revealing programs into seed programs for JVM fuzzing. Based on JavaTailor, Gao et al. [37] further proposed VECT to promote the JVM Testing performance via vectorizing ingredients. Zang et al. [40] proposed JAttack to generate new test programs by filling predefined holes in template classes with domain knowledge for JVM fuzzing. Le et al. [25] proposed Equivalence Modulo Inputs (EMI) to fuzz C compilers, which generates equivalent programs by removing unexecuted code under a set of inputs in seed programs. Donaldson et al. [38] proposed GLFuzz for OpenGL compilers, which designs several mutation strategies to construct equivalent programs. Also, AFL (i.e., American Fuzzy Loop [29]) is a general fuzzer that designs many byte-level and token-level mutation rules to modify seed inputs. In particular, Bang et al. [41] proposed mlir-tv, which applies translation validation [42] on seed programs to verify the correctness of pass implementation.

Our work is orthogonal to this category of techniques, since the generated MLIR programs by MLIRSmith can be used as seed programs for them. In the future, we can explore their integration to further improve the fuzzing effectiveness.

VII. Conclusion

We propose the first technique, MLIRSmith, to fuzz MLIR compiler infrastructure, which is an emerging system benefiting the construction of compilers in various domains. Its core lies in passes (supporting various transformations and optimizations on MLIR programs) and dialects (supporting different levels of abstraction in MLIR). To improve the fuzzing effectiveness, MLIRSmith carefully generates valid and diverse MLIR programs by syntax-guided program template construction and context-aware MLIR program instantiation. By fuzzing the latest revision of MLIR compiler infrastructure for two months, MLIRSmith detected 53 previously unknown bugs, 49/38 of which have been confirmed/fixed by developers.

Acknowledgment

The authors would like to thank anonymous reviewers for their valuable feedback on this paper. We also thank MLIR developers for their technical assistance for this work. This work was supported by the National Natural Science Foundation of China Grant Nos. 62322208, 62002256, 62232001, Key Laboratory of Safety-Critical Software (Nanjing University of Aeronautics and Astronautics), Ministry of Industry and Information Technology Open Project No. NJ2022027, and CCF Young Elite Scientists Sponsorship Program (by CAST).

References

[1] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization, 2004.*

[2] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, M. Cowan, H. Shen, L. Wang, Y. Hu, L. Ceze *et al.*, "Tvm: An automated end-to-end optimizing compiler for deep learning," *arXiv preprint arXiv:1802.04799*, 2018.

[3] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, "MLIR: Scaling compiler infrastructure for domain specific computation," in *2021 IEEE/ACM International Symposium on Code Generation and Optimization, 2004.*

[4] T. D. Le, G. Bercea, T. Chen, A. E. Eichenberger, H. Imai, T. Jin, K. Kawachiya, Y. Negishi, and K. O'Brien, "Compiling ONNX neural network models using MLIR," *CoRR*, 2020.

[5] W. S. Moses, L. Chelini, R. Zhao, and O. Zinenko, in *International Conference on Parallel Architectures and Compilation Techniques, 2021.*

[6] A. Bik, P. Koanantakool, T. Shpeisman, N. Vasilache, B. Zheng, and F. Kjolstad, "Compiler support for sparse tensor computations in mlir," *ACM Transactions on Architecture and Code Optimization, 2022.*

[7] W. S. Moses, I. R. Ivanov, J. Domke, T. Endo, J. Doerfert, and O. Zinenko, "High-performance gpu-to-cpu transpilation and optimization via high-level parallel constructs," in *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming, 2023.*

[8] T. of Bits, "Vast," https://github.com/trailofbits/vast, 2023, gitHub repository.

[9] T. L. Project, "Flang compiler," https://github.com/llvm/llvm-project/tree/main/flang, 2023.

[10] OpenXLA, "Iree," https://openxla.github.io/iree/, 2023.

[11] Q. Shen, H. Ma, J. Chen, Y. Tian, S. Cheung, and X. Chen, "A comprehensive study of deep learning compiler bugs," in *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2021.*

[12] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in C compilers," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, 2011.*

[13] C. Lidbury, A. Lascu, N. Chong, and A. F. Donaldson, "Many-core compiler fuzzing," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2015.

[14] J. Liu, J. Lin, F. Ruffy, C. Tan, J. Li, A. Panda, and L. Zhang, "NNSmith: Generating diverse and valid test cases for deep learning compilers," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, 2023.*

[15] S. Anse, "SQLsmith: A randomized sql query generator," https://github.com/anse1/sqlsmith.

[16] J. Liu, Y. Wei, S. Yang, Y. Deng, and L. Zhang, "Coverage-guided tensor compiler fuzzing with joint ir-pass mutation," *Proceedings of the 2016 ACM SIGPLAN international conference on object-oriented programming, systems, languages, and applications*, 2022.

[17] Y. Zhao, Z. Wang, J. Chen, M. Liu, M. Wu, Y. Zhang, and L. Zhang, "History-driven test program synthesis for JVM testing," in *IEEE/ACM 44th International Conference on Software Engineering, 2022.*

[18] H. Ma, Q. Shen, Y. Tian, J. Chen, and S.-C. Cheung, "Hirfuzz: Detecting high-level optimization bugs in dl compilers via computational graph generation," *arXiv preprint arXiv:2208.02193*, 2022.

[19] H. S. Chae, G. Woo, T. Y. Kim, J. H. Bae, and W.-Y. Kim, "An automated approach to reducing test suites for testing retargeted c compilers for embedded systems," *Journal of Systems and Software*, 2011.

[20] J. Wang, B. Chen, L. Wei, and Y. Liu, "Skyfire: Data-driven seed generation for fuzzing," in *IEEE Symposium on Security and Privacy (SP)*, 2017.

[21] MLIR, "Passes documentation," https://mlir.llvm.org/docs/Dialects/, 2023, gitHub repository.

[22] H. You, Z. Wang, J. Chen, S. Liu, and S. Li, "Regression fuzzing for deep learning systems," in *IEEE/ACM International Conference on Software Engineering*, 2023.

[23] T. D. Nguyen, L. H. Pham, J. Sun, Y. Lin, and Q. T. Minh, "Sfuzz: An efficient adaptive fuzzer for solidity smart contracts," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, 2020*.

[24] H. Wang, Z. Wang, S. Liu, J. Sun, Y. Zhao, Y. Wan, and T. D. Nguyen, "sfuzz2.0: Storage-access pattern guided smart contract fuzzing," *Journal of Software: Evolution and Process, 2023*.

[25] V. Le, M. Afshari, and Z. Su, "Compiler validation via equivalence modulo inputs," in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014.

[26] MLIR, "Passes documentation," https://mlir.llvm.org/docs/Passes/, 2023.

[27] M. Fehr, J. Niu, R. Riddle, M. Amini, Z. Su, and T. Grosser, "Irdl: An ir definition language for ssa compilers," in *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2022.

[28] "Mlirsmith," https://github.com/Colloportus0/MLIRSmith, 2023.

[29] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++ : Combining incremental steps of fuzzing research," in *14th USENIX Workshop on Offensive Technologies*, 2020.

[30] S. A. Chowdhury, S. L. Shrestha, T. T. Johnson, and C. Csallner, "Slemi: Equivalence modulo input (emi) based mutation of cps models for finding compiler bugs in simulink," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020.

[31] V. Le, C. Sun, and Z. Su, "Finding deep compiler bugs via guided stochastic program mutation," in *International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2015.

[32] J. Chen, J. Han, P. Sun, L. Zhang, D. Hao, and L. Zhang, "Compiler bug isolation via effective witness test program generation," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019.

[33] J. Chen, C. Suo, J. Jiang, P. Chen, and X. Li, "Compiler test-program generation via memoized configuration search," in *IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023.

[34] H. Ma, Q. Shen, Y. Tian, J. Chen, and S.-C. Cheung, "Fuzzing deep learning compilers with hirgen," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023.

[35] Y. Zhao, J. Chen, R. Fu, H. Ye, and Z. Wang, "Testing the compiler for a new-born programming language: An industrial case study (experience paper)," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023.

[36] C. Holler, K. Herzig, and A. Zeller, "Fuzzing with code fragments," in *USENIX Security Symposium*, 2012.

[37] T. Gao, J. Chen, Y. Zhao, Y. Zhang, and L. Zhang, "Vectorizing program ingredients for better jvm testing," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023.

[38] A. F. Donaldson, H. Evrard, A. Lascu, and P. Thomson, "Automated testing of graphics shader compilers," 2017.

[39] J. Chen, J. Patra, M. Pradel, Y. Xiong, H. Zhang, D. Hao, and L. Zhang, "A survey of compiler testing," *ACM Comput. Surv.*, 2021.

[40] Z. Zang, N. Wiatrek, M. Gligoric, and A. Shi, "Compiler testing using template java programs," in *IEEE/ACM International Conference on Automated Software Engineering*, 2022.

[41] S. Bang, S. Nam, I. Chun, H. Y. Jhoo, and J. Lee, "Smt-based translation validation for machine learning compiler," in *Computer Aided Verification*, 2022.

[42] A. Pnueli, M. Siegel, and E. Singerman, "Translation validation," in *Tools and Algorithms for the Construction and Analysis of Systems*, 1998.